12 Saving data with Entity Framework Core

This chapter includes

- What Entity Framework Core is and why you should use it
- Adding Entity Framework Core to an ASP.NET Core application
- Building a data model and using it to create a database
- Querying, creating, and updating data using Entity Framework Core

Most applications that you'll build with ASP.NET Core will require storing and loading some kind of data. Even the examples so far in this book have assumed you have some sort of data store—storing exchange rates, user shopping carts, or the locations of physical stores. I've glossed over this for the most part, but typically you'll store this data in a database.

Working with databases can often be a rather cumbersome process. You have to manage connections to the database, translate data from your application to a format the database can understand, and handle a plethora of other subtle issues.

You can manage this complexity in a variety of ways, but I'm going to focus on using a library built primarily for .NET Core and .NET 5.0: Entity Framework Core (EF Core). EF Core is a library that lets you quickly and easily build database access code for your ASP.NET Core applications. It's modeled on the popular Entity Framework 6.x library, but it has significant changes that mean it stands alone in its own right and is more than an upgrade.

The aim of this chapter is to provide a quick overview of EF Core and how you can use it in your applications to quickly query and save to a database. You'll learn enough to connect your app to a database and to manage schema changes to the database, but I won't be going into great depth on any topics.

> **NOTE** For an in-depth look at EF Core, I recommend *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021). Alternatively, you can read about EF Core and its cousin, Entity Framework, on the Microsoft documentation website at https://docs.microsoft.com/ef/core/.

Section 12.1 introduces EF Core and explains why you might want to use it in your applications. You'll learn how the design of EF Core helps you to quickly iterate on your database structure and reduce the friction of interacting with a database.

In section 12.2 you'll learn how to add EF Core to an ASP.NET Core app and configure it using the ASP.NET Core configuration system. You'll see how to build a model for your app that represents the data you'll store in the database and how to hook it into the ASP.NET Core DI container.

> **NOTE** For this chapter and the rest of the book, I'm going to be using SQL Server Express's LocalDB feature. This is installed as part of Visual Studio 2019 (when you choose the ASP.NET and Web Development workload), and it provides a lightweight SQL Server engine.[1] Very little of the code is specific to SQL Server, so you should be able to follow along with a different database if you prefer. The code sample for the book includes a version using SQLite, for example.

No matter how carefully you design your original data model, the time will come when you need to change it. In section 12.3 I show how you can easily update your model and apply these changes to the database itself, using EF Core for all the heavy lifting.

Once you have EF Core configured and a database created, section 12.4 shows how to use EF Core in your application code. You'll see how to create, read, update, and delete (CRUD) records, and you'll learn about some of the patterns to use when designing your data access.

In section 12.5 I highlight a few of the issues you'll want to take into consideration when using EF Core in a production app. A single chapter on EF Core can only offer a brief introduction to all of the related concepts, so if you choose to use EF Core in your

[1] You can read more about LocalDB in Microsoft's "SQL Server Express LocalDB" documentation at http://mng.bz/5jEa.

own applications—especially if this is your first time using such a data access library—I strongly recommend reading more once you have the basics from this chapter.

Before we get into any code, let's look at what EF Core is, what problems it solves, and when you might want to use it.

12.1 Introducing Entity Framework Core

Database access code is ubiquitous across web applications. Whether you're building an e-commerce app, a blog, or the Next Big Thing™, chances are you'll need to interact with a database.

Unfortunately, interacting with databases from app code is often a messy affair, and there are many different approaches you can take. For example, something as simple as reading data from a database requires handling network connections, writing SQL statements, and handling variable result data. The .NET ecosystem has a whole array of libraries you can use for this, ranging from the low-level ADO.NET libraries to higher-level abstractions like EF Core.

In this section, I describe what EF Core is and the problem it's designed to solve. I cover the motivation behind using an abstraction such as EF Core, and how it helps to bridge the gap between your app code and your database. As part of that, I present some of the trade-offs you'll make by using it in your apps, which should help you decide if it's right for your purposes. Finally, we'll take a look at an example EF Core mapping, from app code to database, to get a feel for EF Core's main concepts.

12.1.1 What is EF Core?

EF Core is a library that provides an object-oriented way to access databases. It acts as an *object-relational mapper* (ORM), communicating with the database for you and mapping database responses to .NET classes and objects, as shown in figure 12.1.

> **DEFINITION** With an *object-relational mapper* (ORM), you can manipulate a database using object-oriented concepts such as classes and objects by mapping them to database concepts such as tables and columns.

EF Core is based on, but is distinct from, the existing Entity Framework libraries (currently up to version 6.x). It was built as part of the .NET Core push to work cross-platform, but with additional goals in mind. In particular, the EF Core team wanted to make a highly performant library that could be used with a wide range of databases.

There are many different types of databases, but probably the most commonly used family is *relational* databases, accessed using Structured Query Language (SQL). This is the bread and butter of EF Core; it can map Microsoft SQL Server, MySQL, Postgres, and many other relational databases. It even has a cool in-memory feature you can use when testing, to create a temporary database. EF Core uses a provider model, so that support for other relational databases can be plugged in later, as they become available.

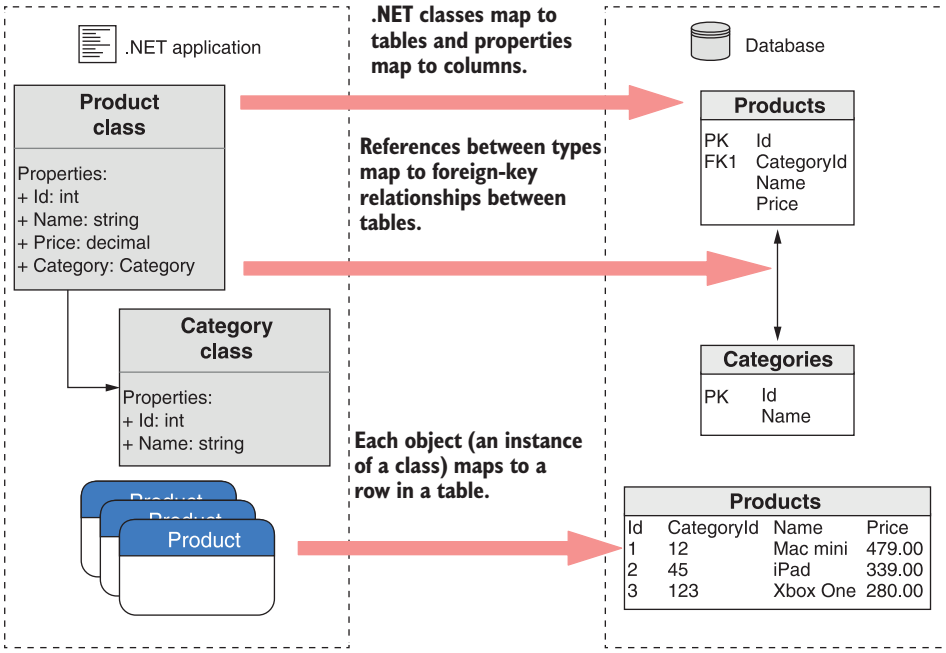

Figure 12.1 EF Core maps .NET classes and objects to database concepts such as tables and rows.

> **NOTE** As of .NET Core 3.0, EF Core now also works with nonrelational, *NoSQL*, or *document* databases like Cosmos DB too. I'm only going to consider mapping to relational databases in this book, however, as that's the most common requirement in my experience. Historically, most data access, especially in the .NET ecosystem, has been using relational databases, so it generally remains the most popular approach.

That covers what EF Core is, but it doesn't dig into why you'd want to use it. Why not access the database directly using the traditional ADO.NET libraries? Most of the arguments for using EF Core can be applied to ORMs in general, so what are the advantages of an ORM?

12.1.2 Why use an object-relational mapper?

One of the biggest advantages an ORM brings is the speed with which you can develop an application. You can stay in the familiar territory of object-oriented .NET, often without ever needing to directly manipulate a database or write custom SQL.

Imagine you have an e-commerce site, and you want to load the details of a product from the database. Using low-level database access code, you'd have to open a connection to the database, write the necessary SQL using the correct table and column names, read the data over the connection, create a POCO to hold the data, and

manually set the properties on the object, converting the data to the correct format as you go. Sounds painful, right?

An ORM, such as EF Core, takes care of most of this for you. It handles the connection to the database, generating the SQL, and mapping data back to your POCO objects. All you need to provide is a LINQ query describing the data you want to retrieve.

ORMs serve as high-level abstractions over databases, so they can significantly reduce the amount of plumbing code you need to write to interact with a database. At the most basic level, they take care of mapping SQL statements to objects and vice versa, but most ORMs take this a step further and provide additional features.

ORMs like EF Core keep track of which properties have changed on any objects they retrieve from the database. This lets you load an object from the database by mapping it from a database table, modify it in .NET code, and then ask the ORM to update the associated record in the database. The ORM will work out which properties have changed and issue update statements for the appropriate columns, saving you a bunch of work.

As is so often the case in software development, using an ORM has its drawbacks. One of the biggest advantages of ORMs is also their Achilles' heel—they hide the database from you. Sometimes this high level of abstraction can lead to problematic database query patterns in your apps. A classic example is the *N*+1 problem, where what should be a single database request turns into separate requests for every single row in a database table.

Another commonly cited drawback is performance. ORMs are abstractions over several concepts, so they inherently do more work than if you were to handcraft every piece of data access in your app. Most ORMs, EF Core included, trade off some degree of performance for ease of development.

That said, if you're aware of the pitfalls of ORMs, you can often drastically simplify the code required to interact with a database. As with anything, if the abstraction works for you, use it; otherwise, don't. If you only have minimal database access requirements or you need the best performance you can get, an ORM such as EF Core may not be the right fit.

An alternative is to get the best of both worlds: use an ORM for the quick development of the bulk of your application, and then fall back to lower-level APIs such as ADO.NET for those few areas that prove to be the bottlenecks in your application. That way, you can get good-enough performance with EF Core, trading off performance for development time, and only optimize those areas that need it.

Even if you do decide to use an ORM in your app, there are many different ORMs available for .NET, of which EF Core is one. Whether EF Core is right for you will depend on the features you need and the trade-offs you're willing to make to get them. The next section compares EF Core to Microsoft's other offering, Entity Framework, but there many other alternatives you could consider, such as Dapper and NHibernate, each with their own set of trade-offs.

12.1.3 When should you choose EF Core?

Microsoft designed EF Core as a reimagining of the mature Entity Framework 6.x (EF 6.x) ORM, which it released in 2008. With ten years of development behind it, EF 6.x is a stable and feature-rich ORM.

In contrast, EF Core is a *comparatively* new project. The APIs of EF Core are designed to be close to those of EF 6.x—though they aren't identical—but the core components have been completely rewritten. You should consider EF Core as distinct from EF 6.x; upgrading directly from EF 6.x to EF Core is nontrivial.

Microsoft supports both EF Core and EF 6.x, and both will see ongoing improvements, so which should you choose? You need to consider a number of things:

- *Cross-platform*—EF Core 5.0 targets .NET Standard, so it can be used in cross-platform apps that target .NET Core 3.0 or later. Since version 6.3, EF 6.x is *also* cross-platform, with some limitations when running on .NET 5.0, such as no designer support.
- *Database providers*—Both EF 6.x and EF Core let you connect to various database types by using pluggable providers. EF Core has a growing number of providers, but there aren't as many for EF 6.x, especially if you want to run EF 6.x on .NET 5.0. If there isn't a provider for the database you're using, that's a bit of a deal breaker!
- *Performance*—The performance of EF 6.x has been a bit of a black mark on its record, so EF Core aims to rectify that. EF Core is designed to be fast and lightweight, significantly outperforming EF 6.x. But it's unlikely to ever reach the performance of a more lightweight ORM, such as Dapper, or handcrafted SQL statements.
- *Features*—Features are where you'll find the biggest disparity between EF 6.x and EF Core, though this difference is smaller with EF Core 5.0 than ever before. EF Core now has many features that EF 6.x doesn't have (batching statements, client-side key generation, in-memory database for testing). EF Core is still missing some features compared to EF 6.x, such as stored procedure mapping and Table-Per-Concrete-Type (TPC), but as EF Core is under active development, these features are on the backlog for implementation.[2] In contrast, EF 6.x will likely only see incremental improvements and bug fixes, rather than major feature additions.

Whether these trade-offs and limitations are a problem for you will depend a lot on your specific app. It's a lot easier to start a new application bearing these limitations in mind than trying to work around them later.

TIP EF Core isn't recommended for everyone, but it's recommended over EF 6.x for new applications. Be sure you understand the trade-offs involved,

[2] For a detailed list of feature differences between EF 6.x and EF Core, see the documentation at http://mng.bz/GxgA.

and keep an eye on the guidance from the EF team here: https://docs.microsoft.com/ef/efcore-and-ef6.

If you're working on a new ASP.NET Core application, you want to use an ORM for rapid development, and you don't require any of the unavailable features, then EF Core is a great contender. It's also supported out of the box by various other subsystems of ASP.NET Core. For instance, in chapter 14 you'll see how to use EF Core with the ASP.NET Core Identity authentication system for managing users in your apps.

Before we get into the nitty-gritty of using EF Core in your app, I'll describe the application we're going to be using as the case study for this chapter. We'll go over the application and database details and how to use EF Core to communicate between the two.

12.1.4 Mapping a database to your application code

EF Core focuses on the communication between an application and a database, so to show it off, we need an application. This chapter uses the example of a simple cooking app that lists recipes and lets you view a recipe's ingredients, as shown in figure 12.2. Users can browse for recipes, add new ones, edit recipes, and delete old ones.

This is obviously a simple application, but it contains all the database interactions you need with its two *entities*: `Recipe` and `Ingredient`.

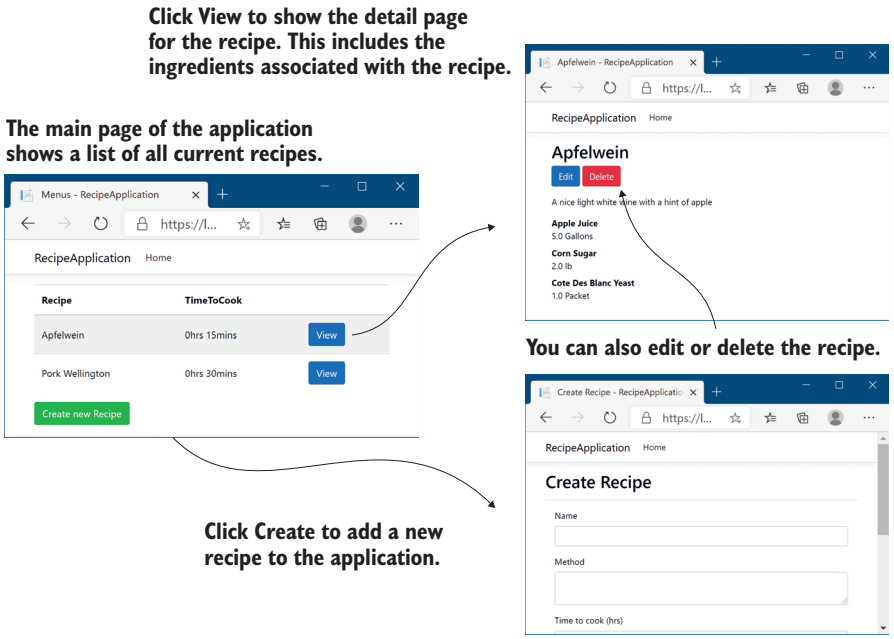

Figure 12.2 The cookery app lists recipes. You can view, update, and delete recipes, or create new ones.

DEFINITION An *entity* is a .NET class that's mapped by EF Core to the database. These are classes you define, typically as POCO classes, that can be saved and loaded by mapping to database tables using EF Core.

When you interact with EF Core, you'll be primarily using POCO entities and a *database context* that inherits from the `DbContext` EF Core class. The entity classes are the object-oriented representations of the tables in your database; they represent the data you want to store in the database. You use the `DbContext` in your application to both configure EF Core and to access the database at runtime.

NOTE You can potentially have multiple `DbContexts` in your application and can even configure them to integrate with different databases.

When your application first uses EF Core, EF Core creates an internal representation of the database, based on the `DbSet<T>` properties on your application's `DbContext` and the entity classes themselves, as shown in figure 12.3.

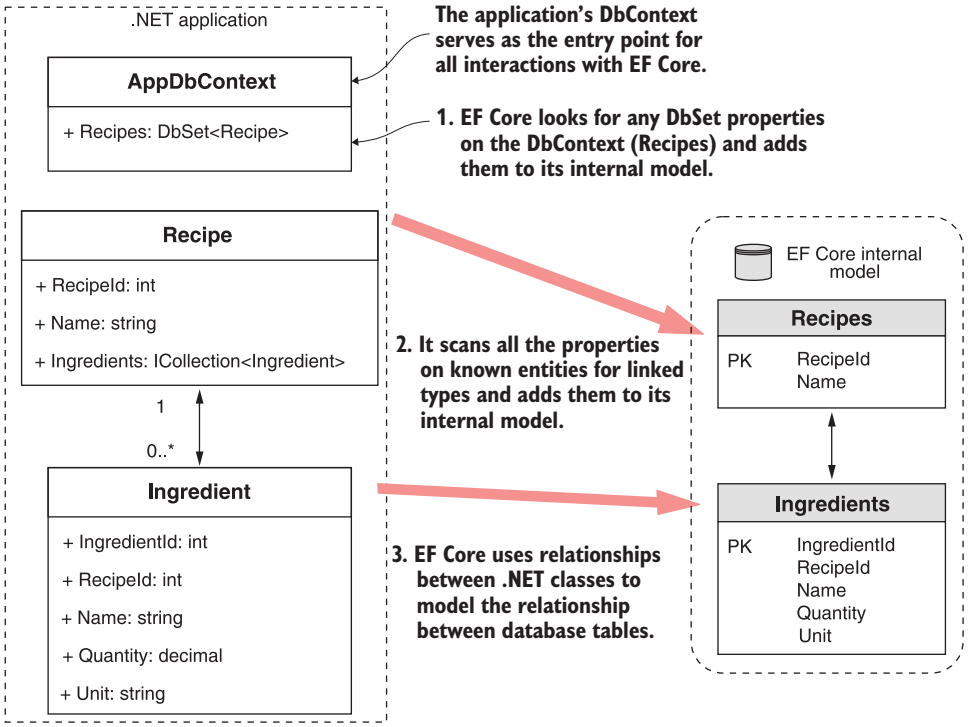

Figure 12.3 EF Core creates an internal model of your application's data model by exploring the types in your code. It adds all of the types referenced in the `DbSet<>` properties on your app's `DbContext`, and any linked types.

For your recipe app, EF Core will build a model of the `Recipe` class because it's exposed on the `AppDbContext` as a `DbSet<Recipe>`. Furthermore, EF Core will loop through all the properties on `Recipe`, looking for types it doesn't know about, and add them to its internal model. In your app, the `Ingredients` collection on `Recipe` exposes the `Ingredient` entity as an `ICollection<Ingredient>`, so EF Core models the entity appropriately.

Each entity is mapped to a table in the database, but EF Core also maps the relationships between the entities. Each recipe can have *many* ingredients, but each ingredient (which has a name, quantity, and unit) belongs to *one* recipe, so this is a many-to-one relationship. EF Core uses that knowledge to correctly model the equivalent many-to-one database structure.

> **NOTE** Two different recipes, say fish pie and lemon chicken, may use an ingredient that has both the same name and quantity, such as the juice of one lemon, but they're fundamentally two different instances. If you update the lemon chicken recipe to use two lemons, you wouldn't want this change to automatically update the fish pie to use two lemons too!

EF Core uses the internal model it builds when interacting with the database. This ensures it builds the correct SQL to create, read, update, and delete entities.

Right, it's about time for some code! In the next section, you'll start building the recipe app. You'll see how to add EF Core to an ASP.NET Core application, configure a database provider, and design your application's data model.

12.2 Adding EF Core to an application

In this section, we'll focus on getting EF Core installed and configured in your ASP.NET Core recipe app. You'll learn how to install the required NuGet packages and how to build the data model for your application. As we're talking about EF Core in this chapter, I'm not going to go into how to create the application in general—I created a simple Razor Pages app as the basis, nothing fancy.

> **TIP** The sample code for this chapter shows the state of the application at three points in this chapter: at the end of section 12.2, at the end of section 12.3, and at the end of the chapter. It also includes examples using both LocalDB and SQLite providers.

Interaction with EF Core in the example app occurs in a service layer that encapsulates all the data access outside of the Razor Pages framework, as shown in figure 12.4. This keeps your concerns separated and makes your services testable.

Adding EF Core to an application is a multistep process:

1. Choose a database provider; for example, Postgres, SQLite, or MS SQL Server.
2. Install the EF Core NuGet packages.
3. Design your app's `DbContext` and entities that make up your data model.
4. Register your app's `DbContext` with the ASP.NET Core DI container.

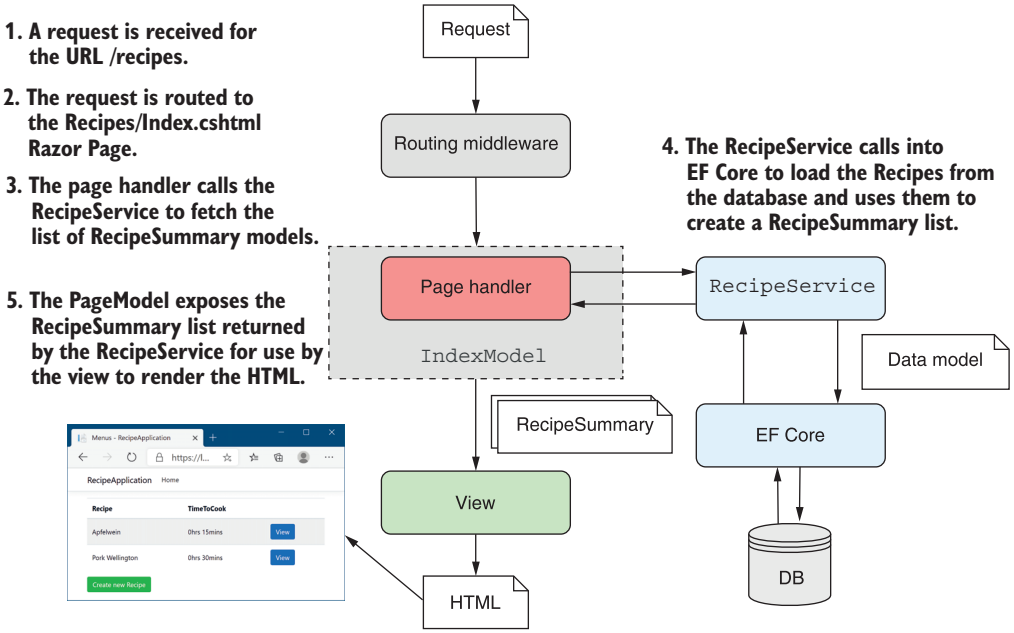

Figure 12.4 Handling a request by loading data from a database using EF Core. Interaction with EF Core is restricted to `RecipeService` only—the Razor Page doesn't access EF Core directly.

5 Use EF Core to generate a *migration* describing your data model.
6 Apply the migration to the database to update the database's schema.

This might seem a little daunting already, but we'll walk through steps 1–4 in this section, and steps 5–6 in section 12.3, so it won't take long. Given the space constraints of this chapter, I'm going to be sticking to the default conventions of EF Core in the code I show. EF Core is far more customizable than it may initially appear, but I encourage you to stick to the defaults wherever possible. It will make your life easier in the long run.

The first step in setting up EF Core is to decide which database you'd like to interact with. It's likely that a client or your company's policy will dictate this to you, but it's still worth giving the choice some thought.

12.2.1 Choosing a database provider and installing EF Core

EF Core supports a range of databases by using a provider model. The modular nature of EF Core means you can use the same high-level API to program against different underlying databases, and EF Core knows how to generate the necessary implementation-specific code and SQL statements.

You probably already have a database in mind when you start your application, and you'll be pleased to know that EF Core has got most of the popular ones covered.

Adding support for a given database involves adding the correct NuGet package to your .csproj file. For example,

- *PostgreSQL*—Npgsql.EntityFrameworkCore.PostgreSQL
- *Microsoft SQL Server*—Microsoft.EntityFrameworkCore.SqlServer
- *MySQL*—MySql.Data.EntityFrameworkCore
- *SQLite*—Microsoft.EntityFrameworkCore.SQLite

Some of the database provider packages are maintained by Microsoft, some are maintained by the open source community, and some may require a paid license (for example, the Oracle provider), so be sure to check your requirements. You can find a list of providers at https://docs.microsoft.com/ef/core/providers/.

You install a database provider into your application in the same way as any other library: by adding a NuGet package to your project's .csproj file and running `dotnet restore` from the command line (or letting Visual Studio automatically restore for you).

EF Core is inherently modular, so you'll need to install multiple packages. I'm using the SQL Server database provider with LocalDB for the recipe app, so I'll be using the SQL Server packages:

- *Microsoft.EntityFrameworkCore.SqlServer*—This is the main database provider package for using EF Core at runtime. It also contains a reference to the main EF Core NuGet package.
- *Microsoft.EntityFrameworkCore.Design*—This contains shared design-time components for EF Core.

TIP You'll also want to install tooling to help you create and update your database. I show how to install these in section 12.3.1.

Listing 12.1 shows the recipe app's .csproj file after adding the EF Core packages. Remember, you add NuGet packages as `PackageReference` elements.

Listing 12.1 Installing EF Core into an ASP.NET Core application

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.SqlServer"
        Version="5.0.0" />
    <PackageReference
        Include="Microsoft.EntityFrameworkCore.Design"
        Version="5.0.0" />
  </ItemGroup>
</Project>
```

The app targets .NET 5.0.

Install the appropriate NuGet package for your selected DB.

Contains shared design-time components for EF Core

With these packages installed and restored, you have everything you need to start building the data model for your application. In the next section we'll create the entity classes and the `DbContext` for your recipe app.

12.2.2 Building a data model

In section 12.1.4, I showed an overview of how EF Core builds up its internal model of your database from the `DbContext` and entity models. Apart from this discovery mechanism, EF Core is pretty flexible in letting you define your entities the way *you* want to, as POCO classes.

Some ORMs require your entities to inherit from a specific base class or you to decorate your models with attributes to describe how to map them. EF Core heavily favors a convention over configuration approach, as you can see in this listing, which shows the `Recipe` and `Ingredient` entity classes for your app.

Listing 12.2 Defining the EF Core entity classes

```
public class Recipe
{
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public string Method { get; set; }
    public ICollection<Ingredient> Ingredients { get; set; }
}
public class Ingredient
{
    public int IngredientId { get; set; }
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public decimal Quantity { get; set; }
    public string Unit { get; set; }
}
```

A Recipe can have many Ingredients, represented by ICollection.

These classes conform to certain default conventions that EF Core uses to build up a picture of the database it's mapping. For example, the `Recipe` class has a `RecipeId` property, and the `Ingredient` class has an `IngredientId` property. EF Core identifies this pattern of an `Id` suffix as indicating the *primary key* of the table.

DEFINITION The *primary key* of a table is a value that uniquely identifies the row among all the others in the table. It's often an `int` or a `Guid`.

Another convention visible here is the `RecipeId` property on the `Ingredient` class. EF Core interprets this to be a *foreign key* pointing to the `Recipe` class. When considered with `ICollection<Ingredient>` on the `Recipe` class, this represents a many-to-one relationship, where each recipe has many ingredients, but each ingredient only belongs to a single recipe, as shown in figure 12.5.

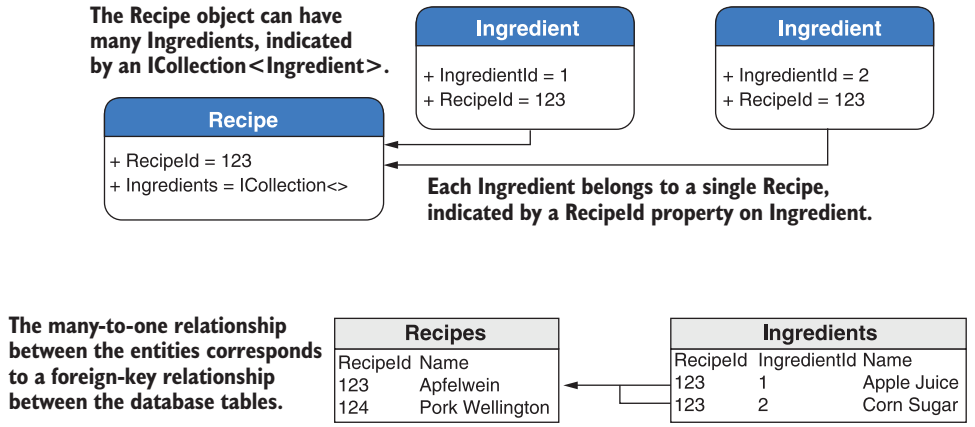

Figure 12.5 Many-to-one relationships in code are translated to foreign-key relationships between tables.

> **DEFINITION** A *foreign key* on a table points to the primary key of a different table, forming a link between the two rows.

Many other conventions are at play here, such as the names EF Core will assume for the database tables and columns, or the database column types it will use for each property, but I'm not going to discuss them here. The EF Core documentation contains details about all of the conventions, as well as how to customize them for your application: https://docs.microsoft.com/ef/core/modeling/.

> **TIP** You can also use `DataAnnotations` attributes to decorate your entity classes, controlling things like column naming or `string` length. EF Core will use these attributes to override the default conventions.

As well as the entities, you also define the `DbContext` for your application. This is the heart of EF Core in your application, used for all your database calls. Create a custom `DbContext`, in this case called `AppDbContext`, and derive from the `DbContext` base class, as shown next. This exposes the `DbSet<Recipe>` so EF Core can discover and map the `Recipe` entity. You can expose multiple instances of `DbSet<>` in this way, for each of the top-level entities in your application.

Listing 12.3 Defining the application `DbContext`

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }
    public DbSet<Recipe> Recipes { get; set; }
}
```

The constructor options object, containing details such as the connection string

You'll use the Recipes property to query the database.

The AppDbContext for your app is simple, containing a list of your root entities, but you can do a lot more with it in a more complex application. If you wanted, you could completely customize how EF Core maps entities to the database, but for this app you're going to use the defaults.

> **NOTE** You didn't list Ingredient on AppDbContext, but it will be modeled by EF Core as it's exposed on the Recipe. You can still access the Ingredient objects in the database, but you must navigate *via* the Recipe entity's Ingredients property to do so, as you'll see in section 12.4.

For this simple example, your data model consists of these three classes: AppDbContext, Recipe, and Ingredient. The two entities will be mapped to tables and their columns to properties, and you'll use the AppDbContext to access them.

> **NOTE** This *code-first* approach is typical, but if you have an existing database, you can automatically generate the EF entities and DbContext instead. (More information can be found in Microsoft's "Reverse Engineering" article in the EF Core documentation: http://mng.bz/mgd4.)

The data model is complete, but you're not quite ready to use it yet. Your ASP.NET Core app doesn't know how to create your AppDbContext, and your AppDbContext needs a connection string so that it can talk to the database. In the next section we'll tackle both of these issues, and we'll finish setting up EF Core in your ASP.NET Core app.

12.2.3 Registering a data context

Like any other service in ASP.Net Core, you should register your AppDbContext with the DI container. When registering your context, you also configure the database provider and set the connection string, so EF Core knows how to talk with the database.

You register the AppDbContext in the ConfigureServices method of Startup.cs. EF Core provides a generic AddDbContext<T> extension method for this purpose, which takes a configuration function for a DbContextOptionsBuilder instance. This builder can be used to set a host of internal properties of EF Core and lets you completely replace the internal services of EF Core if you want.

The configuration for your app is, again, nice and simple, as you can see in the following listing. You set the database provider with the UseSqlServer extension method, made available by the Microsoft.EntityFrameworkCore.SqlServer package, and pass it a connection string.

Listing 12.4 Registering a DbContext with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    var connString = Configuration
        .GetConnectionString("DefaultConnection");
```

The connection string is taken from configuration, from the ConnectionStrings section.

```
    services.AddDbContext<AppDbContext>(
        options => options.UseSqlServer(connString));

    // Add other services.
}
```

Register your app's DbContext by using it as the generic parameter.

Specify the database provider in the customization options for the DbContext.

NOTE If you're using a different database provider, such as a provider for SQLite, you will need to call the appropriate `Use*` method on the `options` object when registering your `AppDbContext`.

The connection string is a typical secret, as I discussed in the previous chapter, so loading it from configuration makes sense. At runtime the correct configuration string for your current environment will be used, so you can use different databases when developing locally and in production.

TIP You can configure your `AppDbContext` in other ways and provide the connection string, such as with the `OnConfiguring` method, but I recommend the method shown here for ASP.NET Core websites.

You now have a `DbContext`, `AppDbContext`, registered with the DI container, and a data model corresponding to your database. Code-wise, you're ready to start using EF Core, but the one thing you *don't* have is a database! In the next section, you'll see how you can easily use the .NET CLI to ensure your database stays up to date with your EF Core data model.

12.3 Managing changes with migrations

In this section you'll learn how to generate SQL statements to keep your database's schema in sync with your application's data model, using migrations. You'll learn how to create an initial migration and use it to create the database. You'll then update your data model, create a second migration, and use it to update the database schema.

Managing *schema* changes for databases, such as when you need to add a new table or a new column, is notoriously difficult. Your application code is explicitly tied to a particular *version* of a database, and you need to make sure the two are always in sync.

DEFINITION *Schema* refers to how the data is organized in a database, including, among others things, the tables, columns, and relationships between them.

When you deploy an app, you can normally delete the old code/executable and replace it with the new code—job done. If you need to roll back a change, delete that new code and deploy an old version of the app.

The difficulty with databases is that they contain data! That means that blowing it away and creating a new database with every deployment isn't possible.

A common best practice is to explicitly version a database's schema along with your application's code. You can do this in a number of ways, but typically you need to store a diff between the previous schema of the database and the new schema, often as a

SQL script. You can then use libraries such as DbUp and FluentMigrator[3] to keep track of which scripts have been applied and ensure your database schema is up to date. Alternatively, you can use external tools to manage this for you.

EF Core provides its own version of schema management called *migrations*. Migrations provide a way to manage changes to a database schema when your EF Core data model changes. A migration is a C# code file in your application that defines how the data model changed—which columns were added, new entities, and so on. Migrations provide a record over time of how your database schema evolved as part of your application, so the schema is always in sync with your app's data model.

You can use command-line tools to create a new database from the migrations, or to update an existing database by *applying* new migrations to it. You can even roll back a migration, which will update a database to a previous schema.

> **WARNING** Applying migrations modifies the database, so you always have to be aware of data loss. If you remove a table from the database using a migration and then roll back the migration, the table will be recreated, but the data it previously contained will be gone forever!

In this section, you'll see how to create your first migration and use it to create a database. You'll then update your data model, create a second migration, and use it to update the database schema.

12.3.1 Creating your first migration

Before you can create migrations, you'll need to install the necessary tooling. There are two primary ways to do this:

- *Package manager console*—You can use PowerShell cmdlets inside Visual Studio's Package Manager Console (PMC). You can install them directly from the PMC or by adding the Microsoft.EntityFrameworkCore.Tools package to your project.
- *.NET tool*—Cross-platform tooling that you can run from the command line and which extends the .NET SDK. You can install these tools globally for your machine by running `dotnet tool install --global dotnet-ef`.[4]

In this book, I'll be using the cross-platform .NET tools, but if you're familiar with EF 6.x or prefer to use the Visual Studio PMC, there are equivalent commands for all of the steps you're going to take.[5] You can check that the .NET tool installed correctly by running `dotnet ef`. This should produce a help screen like the one shown in figure 12.6.

---

[3] DbUp (https://github.com/DbUp/DbUp) and FluentMigrator (https://github.com/fluentmigrator/fluent-migrator) are open source projects.

[4] Alternatively, you can install the tools *locally* using a *tool manifest file*. To see how to use this approach, see my blog article: https://andrewlock.net/new-in-net-core-3-local-tools/.

[5] Documentation for PowerShell cmdlets can be found at https://docs.microsoft.com/ef/core/miscellaneous/cli/powershell.

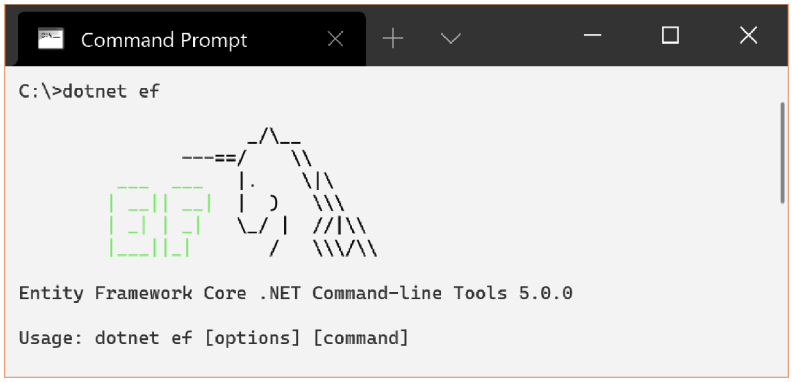

Figure 12.6 Running the `dotnet ef` command to check that the .NET EF Core tools are installed correctly

> **TIP** If you get the "No executable found matching command 'dotnet-ef'" message when running the preceding command, make sure you have installed the global tool using `dotnet tool install --global dotnet-ef`. In general, you need to run the `dotnet ef` tools from the *project* folder in which you have registered your `AppDbContext` (not at the solution-folder level).

With the tools installed and your database context configured, you can create your first migration by running the following command from inside your web project folder and providing a name for the migration—in this case, `InitialSchema`:

```
dotnet ef migrations add InitialSchema
```

This command creates three files in the Migrations folder in your project:

- *Migration file*—A file with the Timestamp_MigrationName.cs format. This describes the actions to take on the database, such as create table or add column. Note that the commands generated here are *database-provider specific*, based on the database provider configured in your project.
- *Migration designer.cs file*—This file describes EF Core's internal model of your data model *at the point in time the migration was generated*.
- *AppDbContextModelSnapshot.cs*—This describes EF Core's *current* internal model. This will be updated when you add another migration, so it should always be the same as the current, latest migration.

  EF Core can use AppDbContextModelSnapshot.cs to determine a database's previous state when creating a new migration, without interacting with the database directly.

These three files encapsulate the migration process, but adding a migration doesn't update anything in the database itself. For that, you must run a different command to apply the migration to the database.

TIP You can, and should, look inside the migration file EF Core generates to check what it will do to your database before running the following commands. Better safe than sorry!

You can apply migrations in one of three ways:

- Using the .NET tool
- Using the Visual Studio PowerShell cmdlets
- In code, by obtaining an instance of your `AppDbContext` from the DI container and calling `context.Database.Migrate()`

Which is best for you depends on how you've designed your application, how you'll update your production database, and your personal preference. I'll use the .NET tool for now, but I discuss some of these considerations in section 12.5.

You can apply migrations to a database by running

```
dotnet ef database update
```

from the project folder of your application. I won't go into the details of how this works, but this command performs four steps:

1. Builds your application.
2. Loads the services configured in your app's `Startup` class, including `AppDbContext`.
3. Checks whether the database in the `AppDbContext` connection string exists. If not, it creates it.
4. Updates the database by applying any unapplied migrations.

If everything is configured correctly, as I showed in section 12.2, then running this command will set you up with a shiny new database, such as the one shown in figure 12.7.

NOTE If you get an error message saying "No project was found" when running these commands, check that you're running them in your application's *project* folder, not the top-level solution folder.

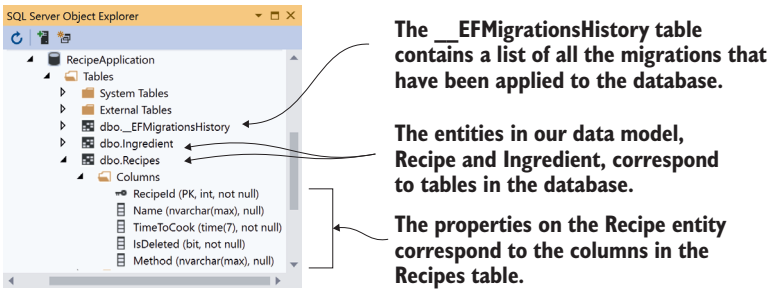

Figure 12.7 Applying migrations to a database will create the database if it doesn't exist and update the database to match EF Core's internal data model. The list of applied migrations is stored in the __EFMigrationsHistory table.

When you apply the migrations to the database, EF Core creates the necessary tables in the database and adds the appropriate columns and keys. You may have also noticed the __EFMigrationsHistory table. EF Core uses this to store the names of migrations that it's applied to the database. Next time you run `dotnet ef database update`, EF Core can compare this table to the list of migrations in your app and will apply only the new ones to your database.

In the next section, we'll look at how this makes it easy to change your data model and update the database schema, without having to recreate the database from scratch.

12.3.2 Adding a second migration

Most applications inevitably evolve, whether due to increased scope or simple maintenance. Adding properties to your entities, adding new entities entirely, and removing obsolete classes—all are likely.

EF Core migrations make this simple. Imagine you decide that you'd like to highlight vegetarian and vegan dishes in your recipe app by exposing `IsVegetarian` and `IsVegan` properties on the `Recipe` entity. Change your entities to your desired state, generate a migration, and apply it to the database, as shown in figure 12.8.

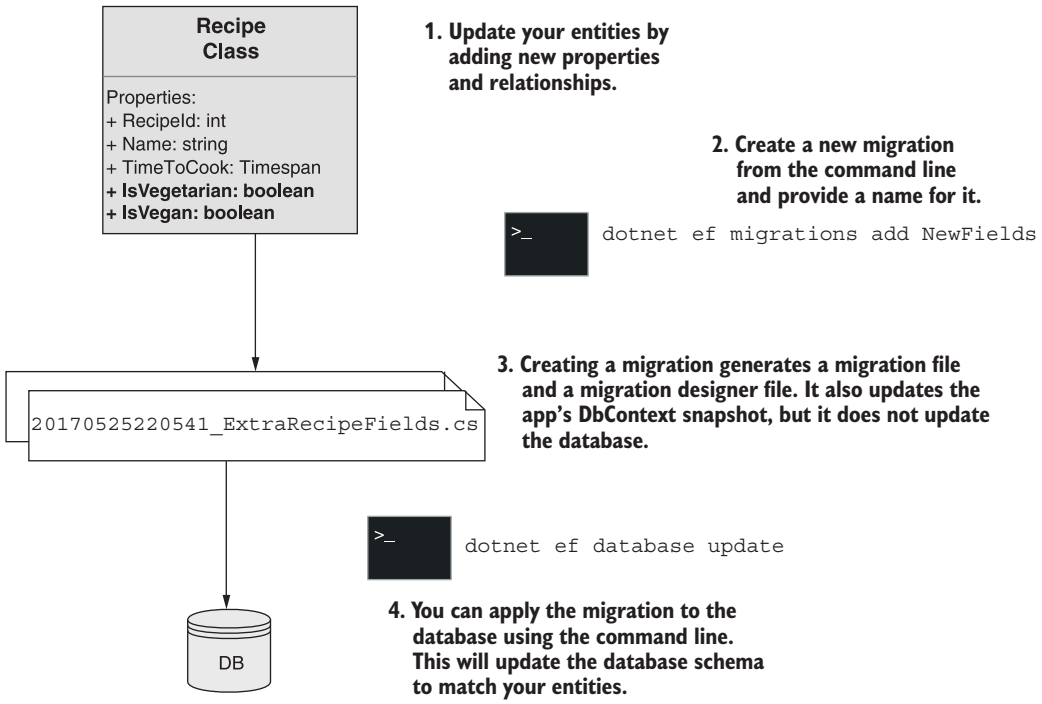

Figure 12.8 Creating a second migration and applying it to the database using the command-line tools.

Listing 12.5 Adding properties to the `Recipe` entity

```
public class Recipe
{
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public TimeSpan TimeToCook { get; set; }
    public bool IsDeleted { get; set; }
    public string Method { get; set; }
    public bool IsVegetarian { get; set; }
    public bool IsVegan { get; set; }
    public ICollection<Ingredient> Ingredients { get; set; }
}
```

After changing your entities, you need to update EF Core's internal representation of your data model. You do this in the exact same way as for the first migration, by calling `dotnet ef migrations add` and providing a name for the migration:

```
dotnet ef migrations add ExtraRecipeFields
```

This creates a second migration in your project by adding the migration file and its .designer.cs snapshot file and updating AppDbContextModelSnapshot.cs, as shown in figure 12.9.

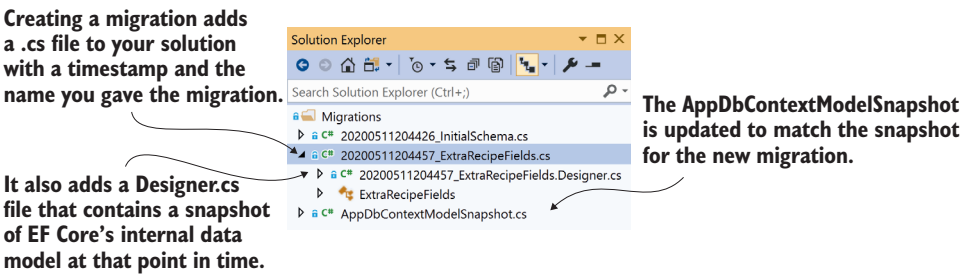

Figure 12.9 Adding a second migration adds a new migration file and a migration Designer.cs file. It also updates AppDbContextModelSnapshot to match the new migration's Designer.cs file.

As before, this creates the migration's files, but it doesn't modify the database. You can apply the migration and update the database by running

```
dotnet ef database update
```

This compares the migrations in your application to the __EFMigrationsHistory table in your database to see which migrations are outstanding, and then it runs them. EF Core will run the 20200511204457_ExtraRecipeFields migration, adding the `IsVegetarian` and `IsVegan` fields to the database, as shown in figure 12.10.

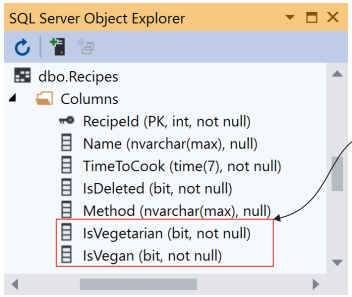

Figure 12.10 Applying the ExtraRecipeFields migration to the database adds the `IsVegetarian` and `IsVegan` fields to the Recipes table.

Using migrations is a great way to ensure your database is versioned along with your app code in source control. You can easily check out your app's source code for a historical point in time and recreate the database schema that your application used at that point.

Migrations are easy to use when you're working alone, or when you're deploying to a single web server, but even in these cases there are important things to consider when deciding how to manage your databases. For apps with multiple web servers using a shared database, or for containerized applications, you have even more things to think about.

This book is about ASP.NET Core, not EF Core, so I don't want to dwell on database management too much, but section 12.5 points out some of the things you need to bear in mind when using migrations in production.

In the next section, we'll get back to the meaty stuff—defining our business logic and performing CRUD operations on the database.

12.4 Querying data from and saving data to the database

Let's review where you are in creating the recipe application:

- You created a simple data model for the application, consisting of recipes and ingredients.
- You generated migrations for the data model, to update EF Core's internal model of your entities.
- You applied the migrations to the database, so its schema matches EF Core's model.

In this section, you'll build the business logic for your application by creating a `RecipeService`. This will handle querying the database for recipes, creating new recipes, and modifying existing ones. As this app only has a simple domain, I'll be using `RecipeService` to handle all the requirements, but in your own apps you may have multiple services that cooperate to provide the business logic.

> **NOTE** For simple apps, you may be tempted to move this logic into your Razor Pages. I'd encourage you to resist this urge; extracting your business logic to other services decouples the HTTP-centric nature of Razor Pages and Web

APIs from the underlying business logic. This will often make your business logic easier to test and more reusable.

Our database doesn't have any data in it yet, so we'd better start by letting you create a recipe.

12.4.1 Creating a record

In this section you're going to build the functionality to let users create a recipe in the app. This will primarily consist of a form that the user can use to enter all the details of the recipe using Razor Tag Helpers, which you learned about in chapters 7 and 8. This form is posted to the Create.cshtml Razor Page, which uses model binding and validation attributes to confirm the request is valid, as you saw in chapter 6.

If the request is valid, the page handler calls `RecipeService` to create the new `Recipe` object in the database. As EF Core is the topic of this chapter, I'm going to focus on this service alone, but you can always check out the source code for this book if you want to see how everything fits together.

The business logic for creating a recipe in this application is simple—there *is* no logic! Map the *command* binding model provided in the Create.cshtml Razor Page to a `Recipe` entity and its `Ingredients`, add the `Recipe` object to `AppDbContext`, and save that in the database, as shown in figure 12.11.

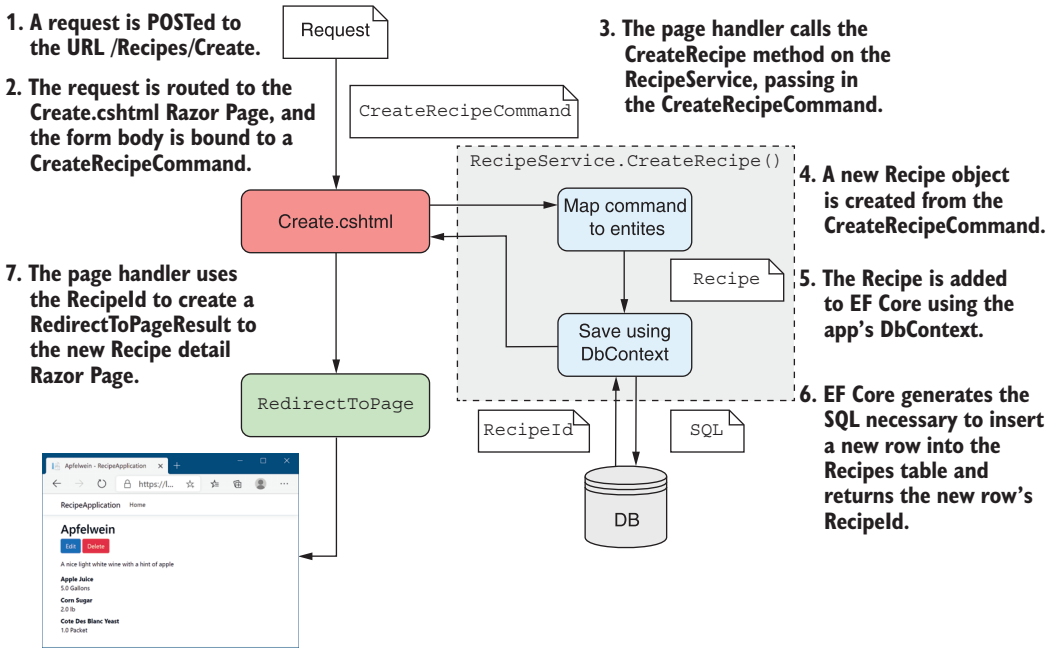

Figure 12.11 Calling the Create.cshtml Razor Page and creating a new entity. A `Recipe` is created from the `CreateRecipeCommand` binding model and is added to the `DbContext`. EF Core generates the SQL to add a new row to the Recipes table in the database.

WARNING Many simple, equivalent sample applications using EF or EF Core allow you to bind *directly* to the `Recipe` entity as the view model for your MVC actions. Unfortunately, this exposes a security vulnerability known as overposting, and it is a bad practice. If you want to avoid the boilerplate mapping code in your applications, consider using a library such as AutoMapper (http://automapper.org/). For more details on overposting, see my blog post on the subject: http://mng.bz/d48O.

Creating an entity in EF Core involves adding a new row to the mapped table. For your application, whenever you create a new `Recipe`, you also add the linked `Ingredient` entities. EF Core takes care of linking these all correctly by creating the correct `RecipeId` for each `Ingredient` in the database.

The bulk of the code required for this example involves translating from `CreateRecipeCommand` to the `Recipe` entity—the interaction with the `AppDbContext` consists of only two methods: `Add()` and `SaveChangesAsync()`.

Listing 12.6 Creating a `Recipe` entity in the database

An instance of the AppDbContext is injected in the class constructor using DI.

CreateRecipeCommand is passed in from the Razor Page handler.

```
readonly AppDbContext _context;
public async Task<int> CreateRecipe(CreateRecipeCommand cmd)
{
    var recipe = new Recipe
    {
        Name = cmd.Name,
        TimeToCook = new TimeSpan(
            cmd.TimeToCookHrs, cmd.TimeToCookMins, 0),
        Method = cmd.Method,
        IsVegetarian = cmd.IsVegetarian,
        IsVegan = cmd.IsVegan,
        Ingredients = cmd.Ingredients?.Select(i =>
        new Ingredient
        {
            Name = i.Name,
            Quantity = i.Quantity,
            Unit = i.Unit,
        }).ToList()
    };
    _context.Add(recipe);
    await _context.SaveChangesAsync();
    return recipe.RecipeId;
}
```

Create a Recipe by mapping from the command object to the Recipe entity.

Map each CreateIngredientCommand onto an Ingredient entity.

Tell EF Core to track the new entities.

Tell EF Core to write the entities to the database. This uses the async version of the command.

EF Core populates the RecipeId field on your new Recipe when it's saved.

All interactions with EF Core and the database start with an instance of `AppDbContext`, which is typically DI-injected via the constructor. Creating a new entity requires three steps:

1. Create the `Recipe` and `Ingredient` entities.
2. Add the entities to EF Core's list of tracked entities using `_context.Add(entity)`.

3 Execute the SQL `INSERT` statements against the database, adding the necessary rows to the `Recipe` and `Ingredient` tables, by calling `_context.SaveChangesAsync()`.

TIP There are *sync* and *async* versions of most of the EF Core commands that involve interacting with the database, such as `SaveChanges()` and `SaveChangesAsync()`. In general, the async versions will allow your app to handle more concurrent connections, so I tend to favor them whenever I can use them.

If there's a problem when EF Core tries to interact with your database—you haven't run the migrations to update the database schema, for example—it will throw an exception. I haven't shown it here, but it's important to handle these in your application so you don't present users with an ugly error page when things go wrong.

Assuming all goes well, EF Core updates all the auto-generated IDs of your entities (`RecipeId` on `Recipe`, and both `RecipeId` and `IngredientId` on `Ingredient`). Return the recipe ID to the Razor Page so it can use it; for example, to redirect to the View Recipe page.

And there you have it—you've created your first entity using EF Core. In the next section we'll look at loading these entities from the database so you can view them in a list.

12.4.2 Loading a list of records

Now that you can create recipes, you need to write the code to view them. Luckily, loading data is simple in EF Core, relying heavily on LINQ methods to control which fields you need. For your app, you'll create a method on `RecipeService` that returns a summary view of a recipe, consisting of the `RecipeId`, `Name`, and `TimeToCook` as a `RecipeSummaryViewModel`, as shown in figure 12.12.

NOTE Creating a view model is technically a UI concern rather than a business logic concern. I'm returning them directly from `RecipeService` here mostly to hammer home that you shouldn't be using EF Core entities directly in your Razor Pages.

The `GetRecipes` method in `RecipeService` is conceptually simple and follows a common pattern for querying an EF Core database, as shown in figure 12.13.

EF Core uses a fluent chain of LINQ commands to define the query to return on the database. The `DbSet<Recipe>` property on `AppDataContext` is an `IQueryable`, so you can use all the usual `Select()` and `Where()` clauses that you would with other `IQueryable` providers. EF Core will convert these into a SQL statement to query the database with when you call an *execute* function such as `ToListAsync()`, `ToArrayAsync()`, `SingleAsync()`, or their non-async brethren.

You can also use the `Select()` extension method to map to objects other than your entities as part of the SQL query. You can use this to efficiently query the database by only fetching the columns you need.

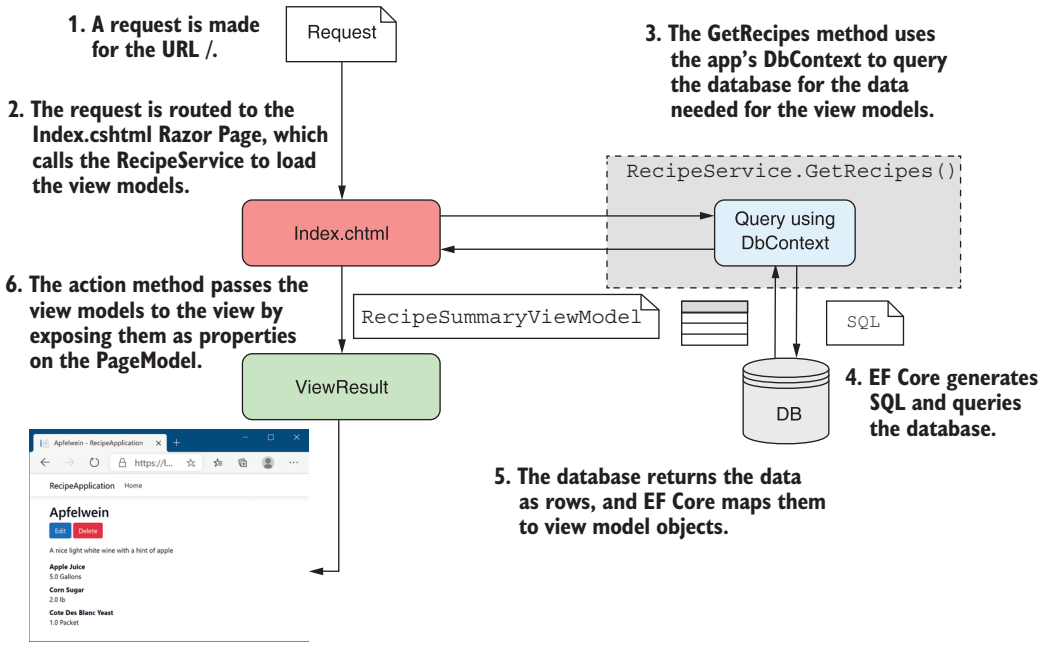

Figure 12.12 Calling the Index.cshtml Razor Page and querying the database to retrieve a list of `RecipeSummaryViewModels`. EF Core generates the SQL to retrieve the necessary fields from the database and maps them to view model objects.

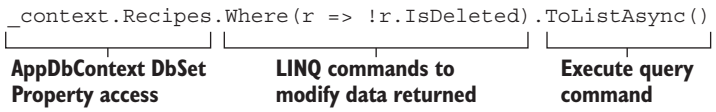

Figure 12.13 The three parts of an EF Core database query

Listing 12.7 shows the code to fetch a list of `RecipeSummaryViewModels`, following the same basic pattern as in figure 12.12. It uses a `Where` LINQ expression to filter out recipes marked as deleted and a `Select` clause to map to the view models. The `ToListAsync()` command instructs EF Core to generate the SQL query, execute it on the database, and build `RecipeSummaryViewModel` from the data returned.

Listing 12.7 Loading a list of items using EF Core in `RecipeService`

```
public async Task<ICollection<RecipeSummaryViewModel>> GetRecipes()
{
    return await _context.Recipes
        .Where(r => !r.IsDeleted)
```

A query starts from a DbSet property.

EF Core will only query the Recipe columns it needs to map the view model correctly.

```
        .Select(r => new RecipeSummaryViewModel
        {
            Id = r.RecipeId,
            Name = r.Name,
            TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"
        })
        .ToListAsync();
}
```

This executes the SQL query and creates the final view models.

Notice that in the `Select` method you convert the `TimeToCook` property from a `TimeSpan` to a `string` using string interpolation:

```
TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"
```

I said before that EF Core converts the series of LINQ expressions into SQL, but that's only a half-truth; EF Core can't or doesn't know how to convert some expressions to SQL. For those cases, such as in this example, EF Core finds the fields from the DB that it needs in order to run the expression on the client side, selects those from the database, and then runs the expression in C# afterwards. This lets you combine the power and performance of database-side evaluation without compromising the functionality of C#.

> **WARNING** Client-side evaluation is both powerful and useful but has the potential to cause issues. In general, recent versions of EF Core will throw an exception if a query requires dangerous client-side evaluation. For examples, including how to avoid these issues, see the documentation at http://mng.bz/zxP6.

At this point, you have a list of records, displaying a summary of the recipe's data, so the obvious next step is to load the detail for a single record.

12.4.3 Loading a single record

For most intents and purposes, loading a single record is the same as loading a list of records. They share the same common structure you saw in figure 12.13, but when loading a single record, you'll typically use a `Where` clause and execute a command that restricts the data to a single entity.

Listing 12.8 shows the code to fetch a recipe by ID, following the same basic pattern as before (figure 12.12). It uses a `Where()` LINQ expression to restrict the query to a single recipe, where `RecipeId == id`, and a `Select` clause to map to `RecipeDetailViewModel`. The `SingleOrDefaultAsync()` clause will cause EF Core to generate the SQL query, execute it on the database, and build the view model.

> **NOTE** `SingleOrDefaultAsync()` will throw an exception if the previous `Where` clause returns more than one record.

Listing 12.8 Loading a single item using EF Core in `RecipeService`

```
public async Task<RecipeDetailViewModel> GetRecipeDetail(int id)
{
    return await _context.Recipes
        .Where(x => x.RecipeId == id)
        .Select(x => new RecipeDetailViewModel
        {
            Id = x.RecipeId,
            Name = x.Name,
            Method = x.Method,
            Ingredients = x.Ingredients
            .Select(item => new RecipeDetailViewModel.Item
            {
                Name = item.Name,
                Quantity = $"{item.Quantity} {item.Unit}"
            })
        })
        .SingleOrDefaultAsync();
}
```

The id of the recipe to load is passed as a parameter.

As before, a query starts from a DbSet property.

Limit the query to the recipe with the provided id.

Map the Recipe to a RecipeDetailViewModel.

Load and map linked Ingredients as part of the same query.

Execute the query and map the data to the view model.

Notice that as well as mapping the `Recipe` to a `RecipeDetailViewModel`, you also map the related `Ingredients` for a `Recipe`, as though you're working with the objects directly in memory. This is one of the advantages of using an ORM—you can easily map child objects and let EF Core decide how best to build the underlying queries to fetch the data.

> **NOTE** EF Core logs all the SQL statements it runs as `LogLevel.Information` events by default, so you can easily see what queries are being run against the database.

Our app is definitely shaping up; you can create new recipes, view them all in a list, and drill down to view individual recipes with their ingredients and method. Pretty soon, though, someone's going to introduce a typo and want to change their data. To do this, you'll have to implement the *U* in CRUD: update.

12.4.4 Updating a model with changes

Updating entities when they have changed is generally the hardest part of CRUD operations, as there are so many variables. Figure 12.14 gives an overview of this process as it applies to your recipe app.

I'm not going to handle the relationship aspect in this book because that's generally a complex problem, and how you tackle it depends on the specifics of your data model. Instead, I'll focus on updating properties on the `Recipe` entity itself.[6]

[6] For a detailed discussion on handling relationship updates in EF Core, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021), chapter 3, section 3.4: http://mng.bz/w9D2.

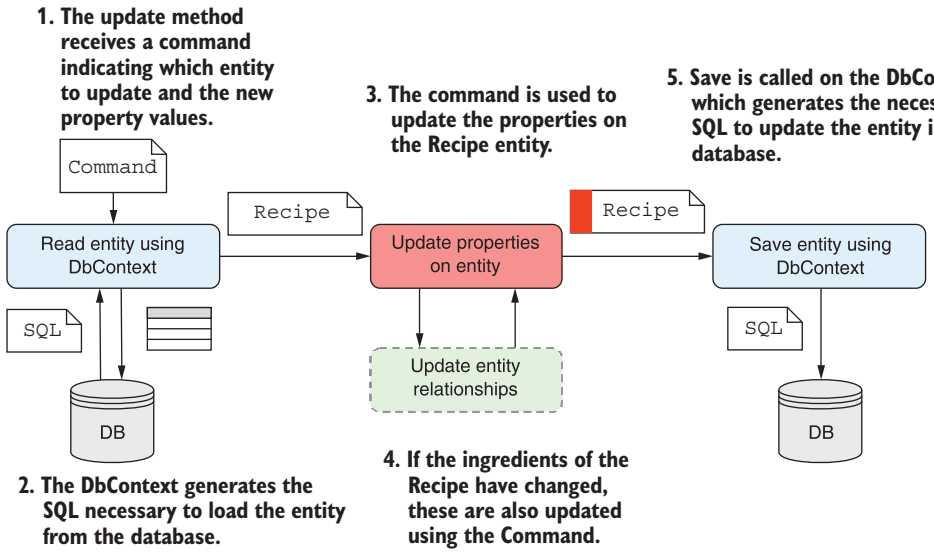

Figure 12.14 Updating an entity involves three steps: read the entity using EF Core, update the properties of the entity, and call `SaveChangesAsync()` on the `DbContext` to generate the SQL to update the correct rows in the database.

For web applications, when you update an entity, you'll typically follow the steps outlined in figure 12.14:

1. Read the entity from the database.
2. Modify the entity's properties.
3. Save the changes to the database.

You'll encapsulate these three steps in a method on `RecipeService` called `UpdateRecipe`. This method takes an `UpdateRecipeCommand` parameter and contains the code to change the `Recipe` entity.

> **NOTE** As with the `Create` command, you don't directly modify the entities in the Razor Page, ensuring you keep the UI concern separate from the business logic.

The following listing shows the `RecipeService.UpdateRecipe` method, which updates the `Recipe` entity. It performs the three steps we defined previously to read, modify, and save the entity. I've extracted the code to update the recipe with the new values to a helper method.

Listing 12.9 Updating an existing entity with EF Core in `RecipeService`

```
public async Task UpdateRecipe(UpdateRecipeCommand cmd)
{
    var recipe = await _context.Recipes.FindAsync(cmd.Id);
```

Find is exposed directly by Recipes and simplifies reading an entity by id.

```
    if(recipe == null) {
        throw new Exception("Unable to find the recipe");
    }
    UpdateRecipe(recipe, cmd);
    await _context.SaveChangesAsync();
}

static void UpdateRecipe(Recipe recipe, UpdateRecipeCommand cmd)
{
    recipe.Name = cmd.Name;
    recipe.TimeToCook =
        new TimeSpan(cmd.TimeToCookHrs, cmd.TimeToCookMins, 0);
    recipe.Method = cmd.Method;
    recipe.IsVegetarian = cmd.IsVegetarian;
    recipe.IsVegan = cmd.IsVegan;
}
```

Set the new values on the Recipe entity.

If an invalid id is provided, recipe will be null.

Execute the SQL to save the changes to the database.

A helper method for setting the new properties on the Recipe entity

In this example I read the `Recipe` entity using the `FindAsync(id)` method exposed by `DbSet`. This is a simple helper method for loading an entity by its ID, in this case `RecipeId`. I could have written a similar query using LINQ as

```
_context.Recipes.Where(r=>r.RecipeId == cmd.Id).FirstOrDefault();
```

Using `FindAsync()` or `Find()` is a little more declarative and concise.

> **TIP** `Find` is actually a bit more complicated. `Find` first checks to see if the entity is already being tracked in EF Core's `DbContext`. If it is (because the entity was previously loaded in this request), the entity is returned immediately without calling the DB. This can obviously be faster if the entity *is* tracked, but it can also be slower if you *know* the entity *isn't* being tracked yet.

You may be wondering how EF Core knows which columns to update when you call `SaveChangesAsync()`. The simplest approach would be to update every column—if the field hasn't changed, then it doesn't matter if you write the same value again. But EF Core is a bit more clever than that.

EF Core internally tracks the *state* of any entities it loads from the database. It creates a snapshot of all the entity's property values, so it can track which ones have changed. When you call `SaveChanges()`, EF Core compares the state of any tracked entities (in this case, the `Recipe` entity) with the tracking snapshot. Any properties that have been changed are included in the `UPDATE` statement sent to the database, and unchanged properties are ignored.

> **NOTE** EF Core provides other mechanisms to track changes, as well as options to disable change-tracking altogether. See the documentation or chapter 3 of Jon P. Smith's *Entity Framework Core in Action*, 2nd ed., (Manning, 2021) for details: http://mng.bz/q9PJ.

With the ability to update recipes, you're almost done with your recipe app. "But wait," I hear you cry, "we haven't handled the *D* in CRUD—delete!" And that's true, but in reality, I've found few occasions when you *want* to delete data.

Let's consider the requirements for deleting a recipe from the application, as shown in figure 12.15. You need to add a (scary-looking) Delete button next to a recipe. After the user clicks Delete, the recipe is no longer visible in the list and can't be viewed.

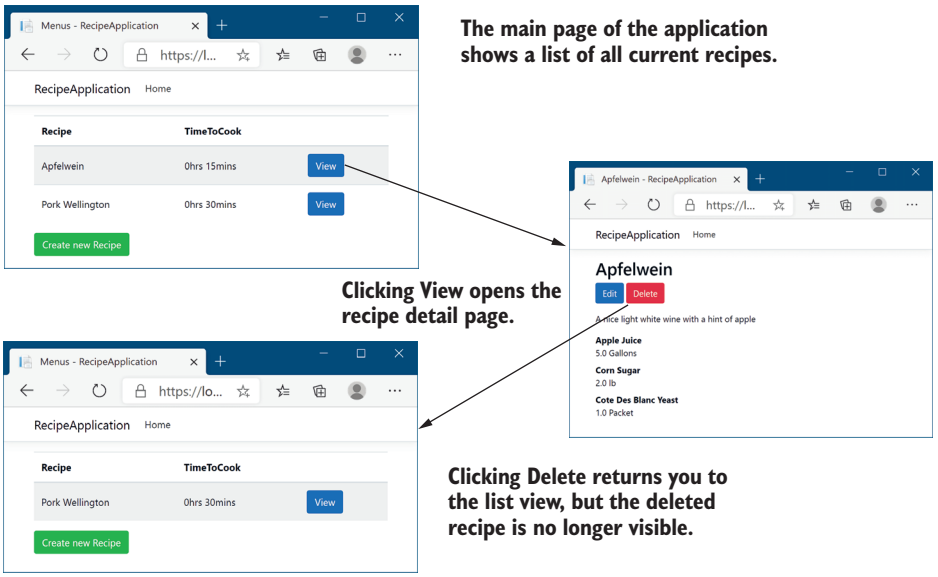

Figure 12.15 The desired behavior when deleting a recipe from the app. Clicking Delete should return you to the application's main list view, with the deleted recipe no longer visible.

You *could* achieve this by deleting the recipe from the database, but the problem with data is that once it's gone, it's gone! What if a user accidentally deletes a record? Also, deleting a row from a relational database typically has implications on other entities. For example, you can't delete a row from the Recipe table in your application without also deleting all the Ingredient rows that reference it, thanks to the foreign-key constraint on `Ingredient.RecipeId`.

EF Core can easily handle these *true deletion* scenarios for you with the `DbContext.Remove(entity)` command, but typically what you *mean* when you find a need to delete data is to "archive" it or hide it from the UI. A common approach to handling this scenario is to include some sort of "Is this entity deleted" flag on your entity, such as the `IsDeleted` flag I included on the `Recipe` entity:

```
public bool IsDeleted {get;set;}
```

If you take this approach, deleting data suddenly becomes simpler, as it's nothing more than an update to the entity. No more issues of lost data, and no more referential integrity problems.

NOTE The main exception I've found to this pattern is when you're storing your users' personally identifying information. In these cases, you may be duty-bound (and, potentially, legally bound) to scrub their information from your database on request.

With this approach, you can create a *delete* method on `RecipeService` that updates the `IsDeleted` flag, as shown in the following listing. In addition, you should ensure you have `Where()` clauses in all the other methods in your `RecipeService`, to ensure you can't display a deleted `Recipe`, as you saw in listing 12.9 for the `GetRecipes()` method.

Listing 12.10 Marking entities as deleted in EF Core

```
public async Task DeleteRecipe(int recipeId)
{
    var recipe = await _context.Recipes.FindAsync(recipeId);
    if(recipe is null) {
        throw new Exception("Unable to find the recipe");
    }
    recipe.IsDeleted = true;
    await _context.SaveChangesAsync();
}
```

Fetch the Recipe entity by id.

If an invalid id is provided, recipe will be null.

Mark the Recipe as deleted.

Execute the SQL to save the changes to the database.

This approach satisfies the requirements—it removes the recipe from the UI of the application—but it simplifies a number of things. This *soft delete* approach won't work for all scenarios, but I've found it to be a common pattern in projects I've worked on.

TIP EF Core has a handy feature called *global query filters*. These allow you to specify a `Where` clause at the model level, so you could, for example, ensure that EF Core *never* loads `Recipes` for which `IsDeleted` is `true`. This is also useful for segregating data in a multi-tenant environment. See the documentation for details: https://docs.microsoft.com/ef/core/querying/filters.

We're almost at the end of this chapter on EF Core. We've covered the basics of adding EF Core to your project and using it to simplify data access, but you'll likely need to learn more about EF Core as your apps become more complex. In the final section of this chapter, I'd like to pinpoint a number of things you need to take into consideration before using EF Core in your own applications, so you're familiar with some of the issues you'll face as your apps grow.

12.5 Using EF Core in production applications

This book is about ASP.NET Core, not EF Core, so I didn't want to spend too much time exploring EF Core. This chapter should've given you enough to get up and running, but you'll definitely need to learn more before you even think about putting EF Core into production. As I've said several times, I recommend *Entity Framework Core in Action,* 2nd ed., by Jon P. Smith (Manning, 2021) for details (http://mng.bz/7Vme) or exploring the EF Core documentation site at https://docs.microsoft.com/ef/core/.

The following topics aren't essential for getting started with EF Core, but you'll quickly come up against them if you build a production-ready app. This section isn't a prescriptive guide to tackling each of these; it's more a set of things to consider before you dive into production.

- *Scaffolding of columns*—EF Core uses conservative values for things like `string` columns by allowing strings of large or unlimited length. In practice, you may want to restrict these and other data types to sensible values.
- *Validation*—You can decorate your entities with `DataAnnotations` validation attributes, but EF Core won't automatically validate the values before saving to the database. This differs from EF 6.x behavior, in which validation was automatic.
- *Handling concurrency*—EF Core provides a few ways to handle concurrency, where multiple users attempt to update an entity at the same time. One partial solution is to use `Timestamp` columns on your entities.
- *Synchronous vs. asynchronous*—EF Core provides both synchronous and asynchronous commands for interacting with the database. Often, async is better for web apps, but there are nuances to this argument that make it impossible to recommend one approach over the other in all situations.

EF Core is a great tool for being productive when writing data-access code, but there are some aspects of working with a database that are unavoidably awkward. The issue of database management is one of the thorniest issues to tackle. This book is about ASP.NET Core, not EF Core, so I don't want to dwell on database management too much. Having said that, most web applications use some sort of database, so the following issues are likely to impact you at some point:

- *Automatic migrations*—If you automatically deploy your app to production as part of some sort of DevOps pipeline, you'll inevitably need some way of applying migrations to a database automatically. You can tackle this in several ways, such as scripting the .NET tool, applying migrations in your app's startup code, or using a custom tool. Each approach has its pros and cons.
- *Multiple web hosts*—One specific consideration is whether you have multiple web servers hosting your app, all pointing to the same database. If so, then applying migrations in your app's startup code becomes harder, as you must ensure only one app can migrate the database at a time.
- *Making backward-compatible schema changes*—A corollary of the multiple web host approach is that you'll often be in a situation where your app is accessing a database that has a *newer* schema than the app thinks. That means you should normally endeavor to make schema changes backward-compatible wherever possible.
- *Storing migrations in a different assembly*—In this chapter I included all my logic in a single project, but in larger apps, data access is often in a different project than the web app. For apps with this structure, you must use slightly different commands when using the .NET CLI or PowerShell cmdlets.

- *Seeding data*—When you first create a database, you often want it to have some initial *seed* data, such as a default user. EF 6.x had a mechanism for seeding data built in, whereas EF Core requires you to explicitly seed your database yourself.

How you choose to handle each of these issues will depend on the infrastructure and deployment approach you take with your application. None of them are particularly fun to tackle, but they're an unfortunate necessity. Take heart, though, they can all be solved one way or another!

That brings us to the end of this chapter on EF Core. In the next chapter we'll look at one of the slightly more advanced topics of MVC and Razor Pages: the filter pipeline, and how you can use it to reduce duplication in your code.

Summary

- EF Core is an object-relational mapper (ORM) that lets you interact with a database by manipulating standard POCO classes, called entities, in your application. This can reduce the amount of SQL and database knowledge you need to have to be productive.
- EF Core maps entity classes to tables, properties on the entity to columns in the tables, and instances of entity objects to rows in these tables. Even if you use EF Core to avoid working with a database directly, you need to keep this mapping in mind.
- EF Core uses a database-provider model that lets you change the underlying database without changing any of your object manipulation code. EF Core has database providers for Microsoft SQL Server, SQLite, PostgreSQL, MySQL, and many others.
- EF Core is cross-platform and has good performance for an ORM, but it has a different feature set than EF 6.x. Nevertheless, EF Core is recommended for all new applications over EF 6.x.
- EF Core stores an internal representation of the entities in your application and how they map to the database, based on the `DbSet<T>` properties on your application's `DbContext`. EF Core builds a model based on the entity classes themselves and any other entities they reference.
- You add EF Core to your app by adding a NuGet database provider package. You should also install the design packages for EF Core. This works in conjunction with the .NET tools to generate and apply migrations to a database.
- EF Core includes many conventions for how entities are defined, such as primary keys and foreign keys. You can customize how entities are defined either declaratively, using `DataAnnotations`, or using a fluent API.
- Your application uses a `DbContext` to interact with EF Core and the database. You register it with a DI container using `AddDbContext<T>`, defining the database provider and providing a connection string. This makes your `DbContext` available in the DI container throughout your app.

- EF Core uses migrations to track changes to your entity definitions. They're used to ensure that your entity definitions, EF Core's internal model, and the database schema all match.
- After changing an entity, you can create a migration either using the .NET tool or using Visual Studio PowerShell cmdlets.
- To create a new migration with the .NET CLI, run `dotnet ef migrations add NAME` in your project folder, where `NAME` is the name you want to give the migration. This compares your current `DbContext` snapshot to the previous version and generates the necessary SQL statements to update your database.
- You can apply the migration to the database using `dotnet ef database update`. This will create the database if it doesn't already exist and apply any outstanding migrations.
- EF Core doesn't interact with the database when it creates migrations, only when you explicitly update the database, so you can still create migrations when you're offline.
- You can add entities to an EF Core database by creating a new entity, `e`, calling `_context.Add(e)` on an instance of your application's data context, `_context`, and calling `_context.SaveChangesAsync()`. This generates the necessary SQL `INSERT` statements to add the new rows to the database.
- You can load records from a database using the `DbSet<T>` properties on your app's `DbContext`. These expose the `IQueryable` interface, so you can use LINQ statements to filter and transform the data in the database before it's returned.
- Updating an entity consists of three steps: reading the entity from the database, modifying the entity, and saving the changes to the database. EF Core will keep track of which properties have changed so that it can optimize the SQL it generates.
- You can delete entities in EF Core using the `Remove` method, but you should consider carefully whether you need this functionality. Often a *soft delete* technique using an `IsDeleted` flag on entities is safer and easier to implement.
- This chapter only covers a subset of the issues you must consider when using EF Core in your application. Before using it in a production app, you should consider, among other things, the data types generated for fields, validation, how to handle concurrency, the seeding of initial data, handling migrations on a running application, and handling migrations in a web-farm scenario.